Tile Architecture and Hardware Implementation for Computation-in-Memory

Mahdi Zahedi, Remon van Duijnen, Stephan Wong, Said Hamdioui
Department of Quantum and Computer Engineering, Delft University of Technology, Delft, The Netherlands
Email: {m.z.zahedi, R.F.J.vanDuijnen, j.s.s.m.wong, s.hamdioui}@tudelft.nl

***Abstract*—Computation-in-memory (CIM) shows great promise for specific applications by employing emerging (non-volatile) memory technologies such as memristors for both storage and compute, greatly reducing energy consumption, and improving performance. Based on our own observations, we can clearly perceive the contours of a generic approach encompassing the use of a memristor array – using technologies such as PCM and ReRAM. In this paper, we present a new instruction-set architecture (ISA) to control a single CIM-tile that comprises the analog memory array itself and all necessary analog and digital periphery. The newly introduced ISA provides the following advantages: (1) flexibility in programming new CIM functionalities by simply rescheduling the instructions from the ISA, (2) definition of a simulation framework, (3) a hardware implementation of the digital periphery, and (4) a design-space exploration of specific CIM-tile operations targeting the aforementioned technologies. For (1), we defined our own compiler that can translate CIM-tile operations to a sequence of instructions from our ISA. The implementation of the digital periphery is synthesized with the 15 nm Nangate library and results regarding power/energy and area are presented. Finally, the design-space exploration is made possible by using the technology-specific parameters with values that have been verified by accurate technology models. All codes of the compiler and simulator as well as the HDL code of the digital periphery are publicly available.**

I. Introduction

Conventional Von Neumann machines inherently separate the processing units from the memory units. The implication of this separation is the need to transfer data back and forth between these units whenever a computation should be performed. The processing speed improvements (due to technological advances) outpaced memory speeds leading to the well-known memory bottleneck [1]. To overcome this, modern computer architectures utilize different approaches such as hierarchical memory architectures, pre-fetching schemes, and parallel computing. However, the speed gap remains and is especially true for big data applications. Furthermore, the energy required to retrieve data from the memory is several orders of magnitude higher than the energy required for a single operation within the processor [2]. Hence, there is a clear need for a new computing paradigm to further advance modern computer architectures considering the aforementioned overheads.

One such paradigm is *computation-in-memory* (CIM) that combats the drawbacks of the Von Neumann architecture by integrating the processing and memory units. Research has shown that certain operations such as Boolean operations and the dot-product are especially susceptible to acceleration using specialized memory arrays, achieving great levels of parallelism and significantly reducing data-traffic and consequently energy consumption [3]. The memory arrays exploit the characteristics of memristive technology to enable the in-memory computations. As Boolean operations and the dot-product can be efficiently supported, specific applications such as database, signal processing, and machine learning can be accelerated using in-memory computing. Recent research in the field of CIM mostly focusses on device characteristics, crossbar structure, and analog periphery circuits to drive and read the crossbar output while a few works are targeting to design a CIM architecture employing memristor devices [4]–[6].

In this paper, we present our detailed CIM-tile architecture which is a generalization of existing designs found in literature and our own work. The accompanying instruction-set architecture (ISA) efficiently controls the data movements to and from the crossbar as well as the digital periphery that is capable of executing additional operations needed for matrix-matrix multiplications. Furthermore, we divided the tile operations into two stages, thereby enabling parallel execution. The introduced ISA bridges the gap between high-level programming languages and the CIM-tile architecture and allows for the definition of a compiler (written in C++) to efficiently schedule the CIM operations instead of the need to devise a complex hardware controller (complex state machine). Summarizing, our main contributions are the following:

- an ISA that is capable to accurately reflect and flexibly control all functional aspects of a generalized CIM-tile organization - targeting different memristor technologies - and, in addition, allow for parallel execution of CIM-tile operations in 2 stages.
- a (SystemC) simulator that is parameterized by incorporating power and timing characteristics for the memristor array taken from detailed technology models. Moreover, different memristors technologies are supported via these parameters. Furthermore, the necessary digital control logic (defined by the ISA) are also incorporated.
- a performance and energy evaluation of two memristor technologies, i.e. PCM and ReRAM, in conjunction with the necessary digital controller and other digital peripheries using our simulator.

II. Background and Related Work

This section provides knowledge on memristive devices and their deployment in computational circuits. Furthermore, prominent CIM architectures and simulators are discussed.

A. Memristive devices and circuits

Memristive devices maintain a relationship between charge and flux elements of a two-terminal passive component. The current-voltage characteristic of a memristive device is a pinched hysteresis loop, as can be seen in Fig 1(a). This means the memristive device can alternate between two stable states by applying a sufficiently high voltage; the high resistance

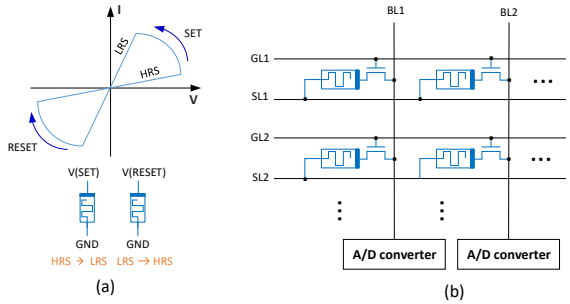

Fig. 1: memristor behavior (a) and 1T1R crossbar structure (b).

state (HRS) or the low-resistance state (LRS), which represent a logic '0' or '1', respectively. In addition to non-volatility, some large benefits of memristor technologies are their overall great scalability, high memory-density, and low leakage power consumption [7]. Examples of such technologies are resistive random access memory (ReRAM) [4] and phase-change-memories (PCMs) [2].

The main focus on memristor-based circuits lies within the use of memristive crossbar array structures, allowing for effective computation of dot product operations [8]. Figure 1(b) portrays the use of memristors in a crossbar array structure. By changing the resistance of each memristive cell within the crossbar array, these memristive memory arrays can exploit Kirchoff and Ohms laws to allow for in-memory computation. The memristor-based computation circuits can be classified into two categories based on the location where the computation result is generated: 1) CIM-Array *(CIM-A)* in which the result is generated within the array [9] 2) CIM-Periphery *(CIM-P)* where the result is obtained in the peripheral circuits [5]. There are some advantages and disadvantages to both classes. Considering CIM-A, while it exploits the entire computation-storage bandwidth, this class suffers from high computation energy and latency as well as reduced device lifetime due to programming the devices for every computation. In the CIM-P class, where the memory-array is only used for data storage, not only the controller can be simplified but also the computation energy, latency, and device lifetime can be improved [8].

B. In-memory architectures and simulators

Considering CIM-A and CIM-P, several architectures have been proposed. ReVAMP [10] (CIM-A) is the first ReRAM-crossbar based general-purpose computing architecture. PRIME [11] and ISAAC [12] (both CIM-P) are ReRAM-based abstract architectures designed to carry out neural network computations in the memory array and produce their results in the read-out circuitry. Similarly, PUMA [13] proposed an accelerator tailored for vector-matrix-multiplication. New high-level in-memory instructions are defined, which are responsible for communicating data between memory units or performing scalar operations in digital peripheries. As CIM architectures consist of many factors influencing the system behavior, simulators are used to be able to explore different configurations and their implications on design metrics such as performance and energy. Accordingly, some memory-oriented simulators for non-volatile cells have been developed such as NVSim [14] and NVmain [15]. Like SPICE, the low-level NVSim simulator provides a circuit-level performance, energy, and area model targeting emerging non-volatile memories. On the other hand, NVMain provides a behavioral memory simulator for architectural level exploration.

With respect to the aforementioned abstract architectures, the proposed tile architecture generalized the existing designs, targeting many operations in the crossbar comprising of normal memory read and write, vector-matrix-multiplication (VMM), and logical operations. This allows independent CIM-tile operation without the need for an external controller knowledgeable of the CIM-tile internals. By the rescheduling of the newly defined in-memory ISA, maximum control over tile can be obtained and flexibly, different patterns of rows and columns can be selected. These patterns are determined according to the **(1) memristor resistance levels, (2) datatype size provided by each application** ('1' and '2' determine how many cells should be allocated to represent a single number), **(3) number of available ADCs, (4) maximum ADC resolution, (5) computation accuracy, and (6) mapping of information to the crossbar**. In addition, considering the aforementioned parameters, **different digital processes** (controlled by ISA) are performed on the crossbar's output (as an intermediate result) to get a meaningful result [16]. Moreover, Contrary to existing aforementioned simulators, our detailed execution model allows us to build a tile-level simulator to bridge the gap between low-level and behavioral-level simulation platforms by actually executing in-memory instructions- after translating high-level kernels to our ISA. Furthermore, the simulator allows the user to track all the control signals and the content of crossbar/registers and produces data dependent energy numbers. Finally, due to the modular programming of the simulator, the user can easily investigate different memristor technologies, circuit designs, and more advanced crossbar modeling (e.g., considering read/write variability).

III. Overall CIM-tile Architecture

In this section, first, we discuss in detail a single CIM-tile architecture and its main components. Secondly, by presenting our in-memory instructions, we explain the decoding structure and how the tile is broken into two stages to achieve better resource utilization and performance improvement.

A. Tile architecture

The overall CIM-tile architecture presented throughout this section is depicted in Figure 2. The CIM-tile is expected to interact with external devices and, therefore, a clear interface need to be defined. The interface comprises two input buffers (Write-Data (WD) and Row-Data (RD)) and an output buffer. These are explained in the following:

- *WD buffer:* This buffer serves as an intermediate storage to alleviate the timing constraints of sending data to the CIM-tile when the (bus) bandwidth is insufficient to transfer all required data in one shot - i.e, to the WD reg(ister). Therefore, each WD buffer row corresponds to a chunk of data received from the outside controller. The WD buffer has been sized such that an entire (array) row of data can

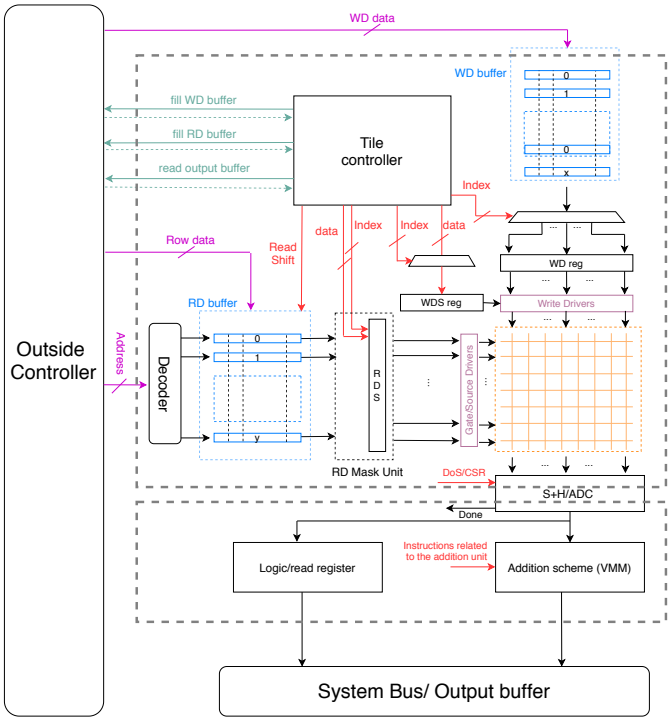

Fig. 2: CIM-tile architecture with 2 stages

be stored inside it. Consequently, after transfer to the WD reg(ister) during a write operation, the WD buffer is ready to receive the next set of data.

- *RD buffer:* As this buffer feeds data into the crossbar bit-by-bit starting from LSB to MSB during a Matrix-Matrix Multiplication (MMM) operation, this buffer is implemented using a parallel-in-serial-out (PISO) register per crossbar row. By having each of the registers sized to fit the maximum supported datatype size (e.g., 32-bit), the buffer only has to be filled once for each of the element dot-product operations. Consequently, after shifting the MSB out of the buffer, the next set of data can be loaded to the buffer to reduce the overhead of data transfer.
- *Output buffer:* This buffer is employed only for temporarily storing the result data until the data has been transferred to other tiles or processors. As the output of the crossbar should be stored in this buffer, its size determines the amount of parallel computation that can be performed.

In addition to the buffers, other digital components (depicted in Figure 2) are:

- *Tile registers:* (i) The *Write Data (WD)* register contains the data that has to be written into the crossbar. The data loads from the *WD* buffer in several steps depending on the WD buffer row size. The register length depends on the width of the crossbar as well as the number of levels supported by the memristor cells. However, due to the flexibility attained by our instructions, we can opt to only partially fill the register if the kernel does not intend to write into all of the columns. (ii) The *Write Data Select (WDS)* register indicates to the write drivers which columns should be written to. The data for this register is embedded in its instruction (see Section III-B) to provide more flexibility for kernels. Finally, (iii) the *Row Data Select (RDS)* register is employed for the activation of crossbar rows and used for all operations including write, read, logic, and VMM. In addition, in the case of VMM, if the ADC resolution does not support the activation of all the rows, this register is used to activate batches of rows in several steps. Similar to the *WDS* register, the data for this register is embedded into its instruction.

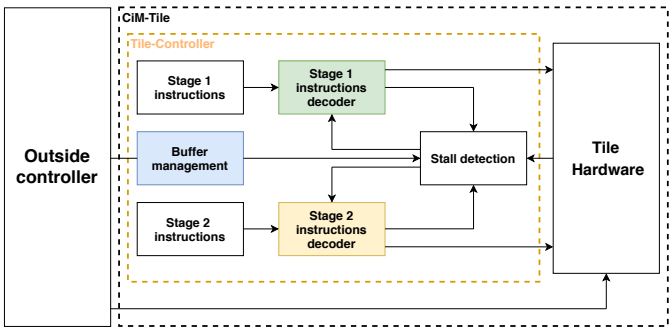

Fig. 3: high-level CIM-tile controller to support 2 stages

- *Tile controller:* Our initial design exploration has shown that, for current technology numbers, breaking the tile into two stages running in parallel, can lead to an improvement in performance and resource utilization. Due to the presence of analog components and unequal number of instructions for each stage, the latency of the stages is not balanced. Therefore, our tile controller has to (i) synchronize these two stages, (ii) decode the instructions to control all the tile components, and (iii) communicate with external devices. The top-level design for the controller is depicted in Figure 3. It comprises two decoders to execute two instructions simultaneously. Each decoder is dedicated to its own stage - i.e., they only decode the instructions of their respective stage. To ensure correct program execution, synchronization between the two decoders is required. This is achieved by monitoring the instructions executed in each stage and stalling the stages when necessary. For example, the state of the analog components is monitored and execution is stalled when the crossbar has not yet finished its current operation. Finally, the controller monitors the states of each of the buffers, setting up flags to the 'outside' when either the input buffers should be filled or the output buffer should be read.
- *Addition unit:* In order to avoid sending intermediate results of MMM to external devices, a new adder structure in the periphery is employed which utilizes minimum-sized adders and is flexible to support different datatype sizes and ADC resolutions. Section III-B explains the related instructions defined to control this unit. More information about the structure of this unit can be found in [16].

B. *In-memory instructions*

To execute a kernel on the tile, a complex sequence of steps have to be carried out considering different operations, the patterns of column and row selections, datatype sizes, and read-out circuitry specifications. A new set of fine-grained in-memory instructions are defined with the objective of keep-

ing the hardware simple and generic while maintaining high flexibility by re-scheduling/programming the instructions and moving the complexity to the compiler. The list of instructions is presented in Table I where the instructions in 'Blue' are executed in the first stage and the rest in the second stage. The instructions are explained shortly in the following.

- Row Data (***RDxx***): The first group of instructions is related to the crossbar rows. *RDSb* instruction provides data for *RDS* register and places its ('Mask data') into the specified index of this register. Due to the bandwidth and instruction size limitation, the register cannot be loaded in one go. In addition, once the *RDS* register has been fully loaded, this instruction only has to be executed when the mask changes again. *RDSc* and *RDSs* instructions reset and set the entire register for extreme patterns of row selections. Finally, *RDsh* will shift the *RD* buffer to the right to present the next bit for a new VMM operation.
- Write Data (***WDxx***): the *WDb* instruction indicates that the data present in the last register of *WD* buffer has to be moved to a section of WD register determined in the instruction by 'index'. Like row selection, *WDSb* instructions load the data to the *WDS* register. Similarly, *WDSc* and *WDSs* instructions reset and set the register respectively, which again is beneficial for some patterns of columns selection.
- Crossbar (***FS, DoA***): To operate on the crossbar, first we need to configure the drivers to provide proper voltage levels. This is done by using the *FS* instruction. This instruction is also used to configure the read-out circuitry as well as bypassing the *RD* buffer data in the case of read/write/logical operations. By decoding the *DoA* instruction, drivers are activated and the operation is started on the crossbar.
- Read Out (***DoS, CS, DoR***): Due to the overhead of Analog-to-Digital Converters (ADCs), they need to be shared between several columns, which translates to the necessity for a *Sample-and-Hold* unit to save the crossbar's output. The *DoS* instruction activates this unit at the right time. The data can be sampled when the second stage is already done with the prior sampled data. The *CS* instruction, which is performed in the second stage, not only indicates which of the ADCs input columns should be selected, but also an activation bit per ADC is used to allow for de-activating an ADC for certain columns which should not be read. The *CS* instruction can be executed only when new data is sampled, otherwise the second stage is stalled.
- Jump instructions (***jal, jr, BNE***): As the read stage often performs identical sets of instructions (e.g., when performing a MMM, the same set of columns has to be read for every dot-product operation), a jump instruction is introduced to save a large portion of the instruction file size. Similarly to MIPS, jump-and-link (*jal*) and jump-register (*jr*) instructions are introduced, allowing for re-using the same block of instructions for every read.
- Addition unit (***LS, IADD, CP, AS, CB***): According to the structure of the addition unit proposed in [16], in the case of low ADC resolution, the *LS* instruction indicates the last section of rows (multiplier) that are activated. The *IADD* instruction performs an addition between different bits of the multiplier. If a number is distributed over more than one ADC, *AS* instructions carry out the addition between the results taken per ADC. Finally, *CP* and *CB* load the result to the output buffer obtained either per ADC or between ADCs, respectively.

TABLE I: overview of the newly proposed ISA

Opcode	Op 1	Op 2	Function description
RDSb	Index	Mask	Place 'Mask' into RDS reg at 'Index'
RDSc			Clear the entire RDS register
RDSs			Set the entire RDS register
RDsh			Shift RD buffer contents
WDb	Index		Copy data to for WD buffer to WD reg. at 'Index'
WDSb	Index	Mask	Place 'Mask' into WDS reg at 'Index'
WDSc			Clear the entire WDS register
WDSs			Set the entire WDS register
FS	Function		Select crossbar functionality
DoA			Activate the crossbar function
DoS			Sample the crossbar output
CS	Index	Activation	Select column to be read by ADC
DoR			Activate the ADC
jal	Address		Jump to 'Address' and store PC.
jr			Jump to the PC stored in return reg.
BNE			Branch to PC stored in branch reg.
LS			Indicate the last section of rows to reads
IADD			Activate third stage of the addition unit
CP			Copy result per ADC to output buffer
AS	Selection		Select adders for addition between ADCs
CB			Copy the result of addition between ADC to output buffer

IV. CIM SIMULATOR AND COMPILER

The compiler is composed of two abstraction levels. First, the high-level compiler presented in [17] extracts the kernels that can be executed on CIM-tile. Second, the low-level compiler translates these kernels into the sequences of aforementioned in-memory instructions. The compiler is aware of the tile configuration and constraints meaning different sequences of instructions are generated for each. To be able to explore different concepts of the CIM-tile architecture, a CIM-tile simulator has been developed. This simulator is written in *SystemC* including TLM interfaces to be able to accurately simulate hardware as it is close to the HDL implementation. The simulator provides a basis which can be altered and expanded upon to allow for initial testing and exploring of solutions proposed throughout this paper. The simulator provides numbers for performance and energy consumption. Furthermore, the waveform to track of the CIM-tile control signals, the content of crossbar cells, and the output data are reported, which are essential for functional testing. Moreover, since it was written in a modular manner, additional functionality such as noise and variation for analog components of the tile can be integrated easily.

The ability of the simulator to actually execute the applications' kernels thanks to the detailed execution model and in-memory instructions, resulted in precise execution time as well as a data-dependent energy number for the crossbar. Equations 1 to 4 are used to calculate the energy and power consumption of the crossbar for reading/computing and writing. R_{rc} represents the resistance value of the cell located at row 'r' and column 'c' in the crossbar. $P_{DIM_{read/write}}$ correspond to the power consumed by the drivers (Digital Input Modular) for reading and writing, respectively. The parameter $activation_{r/c}$ is a binary value indicating whether a row or column has been activated or not. $V_{(read/write)}$ is the crossbar read or write voltage and $I_{(write)}$ is the crossbar write current. Finally, T_{Xbar} is the latency of the crossbar concerning different operations.

TABLE II: value of parameters used for the analog components

Component	Parameters	Spec	
Memristive devices		ReRAM	PCM
	cell levels	2	2
	LRS	5k	20k
	HRS	1M	10M
	read voltage	0.2V	0.2V
	write voltage	2V	1V
	write current	100 μA	300 μA
	read time	10 ns	10 ns
	write time	100 ns	100 ns
Crossbar	structure	1T1R	
	num. columns	256	
	num. rows	256	
S&H	number	256	
	hold time	9.2 ms	
	latching energy	0.25 pJ	
	latency	0.6 ns	
DIM		read DIM	write DIM
	number	256	256
	power	3.9 μW	3.9 μW
ADC	power	2.6 mW	
	precision	8 bits	
	latency	1.2 GSps	

$$P_{(read,compute)} = \sum_{r=1}^{\#rows} ([\sum_{c=1}^{\#columns} (\frac{V_{(read)}^2}{R_{rc}}) + P_{DIM_{read}}] * activation_r) \quad (1)$$

$$P_{(write)} = \sum_{c=1}^{\#columns} [V_{(write)} * I_{(write)} + P_{DIM_{write}}] * activation_c \quad (2)$$

$$E_{(read,compute)} = P_{(read,compute)} * T_{Xbar(read,compute)} \quad (3)$$

$$E_{(write)} = P_{(write)} * T_{Xbar(write)} \quad (4)$$

V. Evaluation and discussion

In this section, the proposed architecture is evaluated in terms of power, energy, and area, and is compared with analog components. As a benchmark, the linear-algebra kernel "GEMM" from the Polybench benchmark suite was chosen. In this kernel, first, the multiplicands are written into the crossbar (write operation) and then the actual multiplication is performed.

A. Simulation setup

The values regarding the CIM-tile analog components are summarized in Table II. The parameters for ReRAM and PCM technologies are taken from [4] and [18] validated for actual devices. The same ADC values used in [12] are considered for our setups. To obtain the power consumption and area for the CIM-tile digital circuits, they were synthesized in Cadence Genus targeting standard cell 15 nm Nangate library. Since all the information and control signals can be tracked by employing our simulator, a typical **activity factor** and performance number are extracted. These numbers are incorporated to obtain accurate energy consumption for the digital as well as analog components of the tile. Finally, functional testing of the architecture is performed on Xilinx Zynq ZC702 board where analog components are modeled with digital circuits. In this side experiment, correct execution of the instructions and the generated final output were evaluated to validate the functionality of the architecture.

B. Simulation results

Despite the abstract architectures mentioned in Section II, in this section, detail information regarding a single tile and its main components are presented. Considering our CIM-tile architecture demonstrated in Figure 2, digital circuits have different contributions to the power consumption. Figure 4(a), depicted for 8-bit maximum supported datatype size in the tile, gives an insight into how much power each of these circuits consumes (8-bit datatype size means that in the case of MMM, each element of multiplicand is distributed over 8 cells and each element of multiplier is fed to the crossbar over 8 time steps). As we expected, the *RD* buffer due to its size (feeding 8-bit data to 256 crossbar rows) and *addition unit* due to its high number of instances consume more power than others. However, since the buffer is not always switching (dynamic power), the average power is much less. The imposed power consumption on the system is the cost of the flexibility of row selection brought by *RD* buffer and its associate instructions.

The overhead of digital circuits in terms of energy is depicted in Figure 4 (b) and (c) to quantify the expense of high flexibility for programming the tile. In this figure, the overhead of the digital parts of our tile is compared with the analog parts. The comparison was performed for both ReRAM and PCM technologies with 256*256 crossbar size considering different datatype sizes. **First**, it is observed that as the datatype size increases, due to more computations, the overall energy is increasing as well. **Second**, by increasing the number of computations, the overhead of crossbar **pr**ogramming reduces (see striped blue bar) since more computations are performed before reprogramming the crossbar. **Third**, digital circuits impose more overhead while moving from 16-bit to 32-bit datatype size (see orange bars as well as energy per VMM). As the datatype size increases, fewer adders and registers need to be instantiated (fewer numbers produced per crossbar activation). These units are essential to accomplish digital post-processing on the crossbar's output to deliver the final result. However, as the data type size increased, each of these adders and registers individually is of a larger size (see [16]). Accordingly, although this customized unit utilizes minimum-size adders and registers, supporting larger datatype size resulted in more overhead for this unit. **Fourth**, the overhead of the digital circuit for the PCM case is higher compared to ReRAM since the PCM crossbar consumes less energy due to its higher value of low resistance state. While the device level researchers are working on devices with a higher value of low resistance, this graph gives a good insight into how much this value contributes to the energy consumption of the system.

Finally, the area comparison of digital and analog circuits is depicted in Figure 5. For this experiment, the area of each cell in 1T1R crossbar structure is taken from [19], where it was fabricated with 22 nm technology. Although our digital circuits were synthesized with 15 nm technology, the result shows their area is around 6 times less than the analog counterpart regardless of the crossbar dimension.

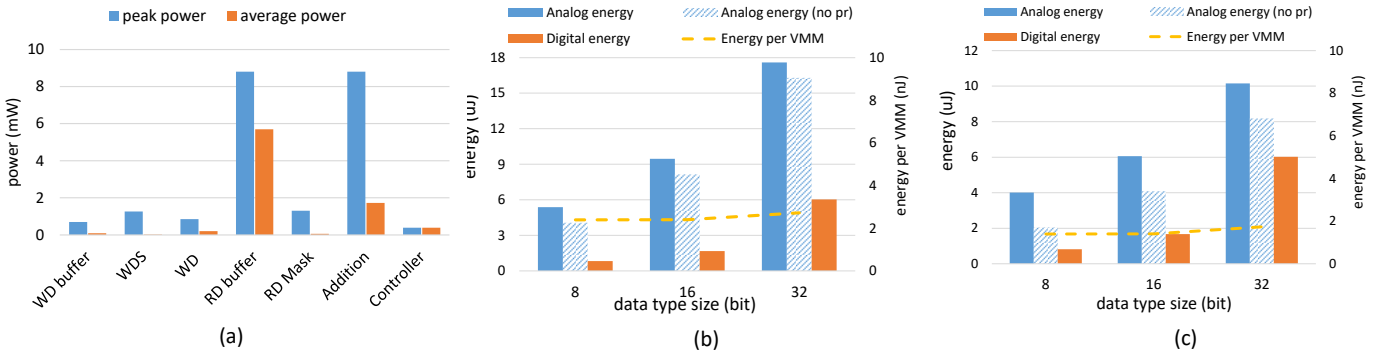

Fig. 4: (a) power break down of digital circuits (b) energy number for different datatype sizes considering ReRAM and (c) PCM

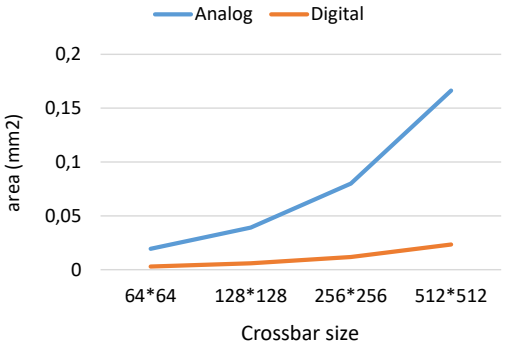

Fig. 5: Area comparison of digital/analog circuits for ReRAM

VI. Conclusion

In this paper, we proposed our programmable CIM-tile architecture for which, by exploiting our ISA, high flexibility are achieved. In addition, the architecture provides a clear interface and independence from external devices. Considering different CIM-tile configurations, we developed our compiler and simulator to facilitate design space exploration. By synthesizing the digital part of the CIM-tile, its overhead compared to the analog counterpart was demonstrated in terms of power, energy, and area. The result will give a remarkable insight into future researches to realize which customizations have to be applied for different configurations and application requirements.

References

[1] S. Srikanth, L. Subramanian, S. Subramoney, T. M. Conte, and H. Wang, "Tackling Memory Access Latency through DRAM Row Management," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. New York, NY, USA: ACM, 2018, pp. 137–147.

[2] M. L. Gallo and A. Sebastian, "An Overview of Phase-change Memory Device Physics," *Journal of Physics D: Applied Physics*, vol. 53, no. 21, p. 213002, mar 2020.

[3] J. Yu, M. A. Lebdeh, H. A. Du Nguyen, M. Taouil, and S. Hamdioui, "The Power of Computation-in-Memory Based on Memristive Devices," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2020, pp. 385–392.

[4] A. Hardtdegen, C. La Torre, F. Cüppers, S. Menzel, R. Waser, and S. Hoffmann-Eifert, "Improved Switching Stability and The Effect of an Internal Series Resistor in HfO 2/TiO x Bilayer ReRAM Cells," *IEEE Transactions on Electron Devices*, vol. 65, no. 8, pp. 3229–3236, 2018.

[5] L. Xie, H. A. Du Nguyen, J. Yu, A. Kaichouhi, M. Taouil, M. AlFailakawi, and S. Hamdioui, "Scouting Logic: A Novel Memristor-based Logic Design for Resistive Computing," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2017, pp. 176–181.

[6] S. Hamdioui, H. A. Du Nguyen, M. Taouil, A. Sebastian, M. Le Gallo, S. Pande, S. Schaafsma, F. Catthoor, S. Das, F. G. Redondo *et al.*, "Applications of Computation-In-Memory Architectures based on Memristive Devices," *DATE*, pp. 486–491, November 2019.

[7] B. Govoreanu, G. Kar, Y. Chen, V. Paraschiv *et al.*, "10×10nm2 Hf/HfOx Crossbar Resistive RAM with Excellent Performance, Reliability and Low-Energy Operation," in *2011 International Electron Devices Meeting*, Dec 2011, pp. 31.6.1–31.6.4.

[8] M. Lebdeh, U. Reinsalu, H. Nguyen, S. Wong, and S. Hamdioui, "Memristive Device Based Circuits for Computation-in-Memory Architectures," *ISCAS*, 2019.

[9] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "Memristive Switches Enable Stateful Logic Operations via Material Implication," *Nature*, vol. 464, no. 7290, p. 873, 2010.

[10] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "ReVAMP: ReRAM Based VLIW Architecture for In-Memory Computing," *Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE*, pp. 782–787, 2017.

[11] C. Ping, L. Shuangchen, X. Cong, T. Zhang, Z. Jishen, L. Yongpan, W. Yu, and X. Yuan, "PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory," *ACM/IEEE*, pp. 27–39, 2016.

[12] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analo Arithmetic in Crossbars," *ACM/IEEE ISCA*, June 2016.

[13] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy *et al.*, "PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 715–731.

[14] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, July 2012.

[15] M. Poremba and Y. Xie, "NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories," in *2012 IEEE Computer Society Annual Symposium on VLSI*, Aug 2012, pp. 392–397.

[16] M. Zahedi, M. Mayahinia, M. A. Lebdeh, S. Wong, and S. Hamdioui, "Efficient Organization of Digital Periphery to Support Integer Datatype for Memristor-Based CIM," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2020, pp. 216–221.

[17] A. Drebes, L. Chelini, O. Zinenko, A. Cohen, H. Corporaal, T. Grosser, K. Vadivel, and N. Vasilache, "TC-CIM: Empowering Tensor Comprehensions for Computing-In-Memory," in *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[18] M. Le Gallo, A. Sebastian, G. Cherubini, H. Giefers, and E. Eleftheriou, "Compressed Sensing with Approximate Message Passing Using In-Memory Computing," *IEEE Transactions on Electron Devices*, vol. 65, no. 10, pp. 4304–4312, 2018.

[19] O. Golonzka, U. Arslan, P. Bai, M. Bohr, O. Baykan, Y. Chang, A. Chaudhari, A. Chen, J. Clarke, C. Connor *et al.*, "Non-volatile RRAM Embedded into 22FFL FinFET Technology," in *2019 Symposium on VLSI Technology*. IEEE, 2019, pp. T230–T231.